



RTX Real-Time Kernel Demonstration

RealView Microcontroller Development Kit

Overview

Aim

To demonstrate with a practical example how the RTX Real-Time Kernel works and how to write applications based on it. The workshop demonstrates some of RTX's key benefits:

- Ease of Use
- Versatility
- Small size
- Configuration and debug support integrated in the Microcontroller Development Kit (MDK)

Requirements

- RealView MDK v3.15 or higher

The software example used in the workshop was written for a Keil MCBSTM32 Evaluation Board. The images in this document show the example running in simulation, but customers can also run it on real hardware.

Note: This demo works with the evaluation version of MDK. When running the evaluation version it displays a warning message that the maximum code size of the project is limited to 16KByte.

Note: This demo assumes basic knowledge of MDK. The "4-step" demo for MCBSTM32E is a good starting point for customers using MDK for the first time.

Conventions



Box shows actions for demonstrator or user of demo.
Icon represents the button controls to be used.

Set-up

Before running the demo for the first time.



Make a copy of the folder
../Keil/ARM/Boards/Keil/MCBSTM32/RTX_Traffic
Into a new demo folder
../Keil/ARM/Boards/Keil/MCBSTM32/Demo

Project Settings

Open and Clear MDK



Project > Close Project

Open an Example Project

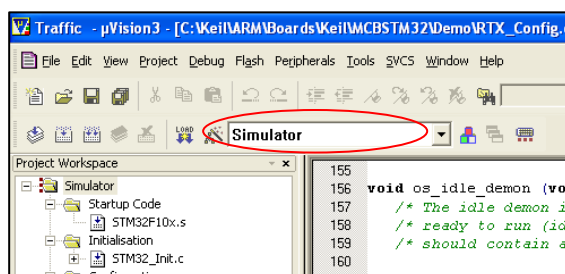


Project > Open Project...
Navigate to ../Keil/ARM/Boards/Keil/MCBSTM32/Demo/Traffic.Uv2

Project Target



Select the Simulator project target



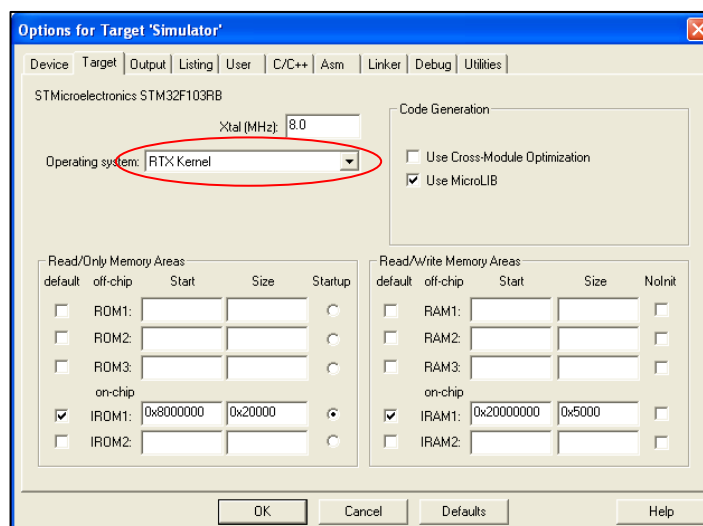
In the RTX_Traffic example the Simulator target is pre-configured to work with RTX and to run the code on a simulated STM32 microcontroller.

Target Options

The μ Vision project must include information about the operating system used by the application, so that the source code is linked with the RTOS libraries. This is specified in the 'Options for Target' dialog.



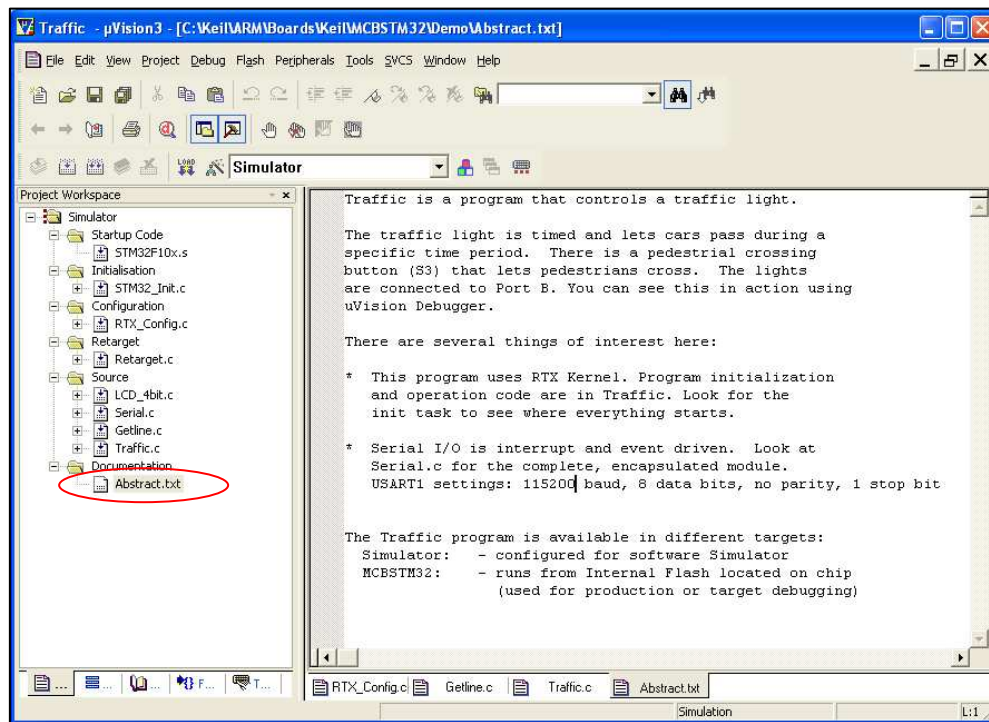
Click 'Options for Target' button
Click on the 'Target' Tab



RTX_Traffic Example

RTX_Traffic Abstract

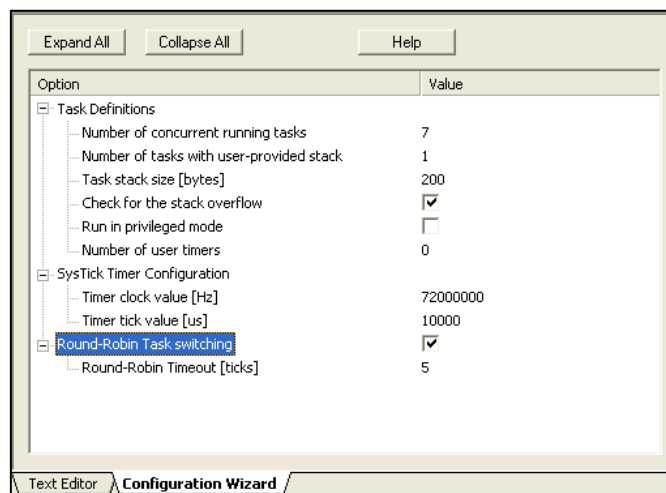
The file Abstract.txt describes the basic functionality of the RTX_Traffic example. Double click on its icon to open it.



RTX_Traffic Source Files

The following source files include code to configure and make use of the RTX functionality.

RTX_Config.c is part of RTX. It allows you to configure parameters such as maximum number of concurrent tasks, stack size for each task, period of system timer and type of scheduling. All these options can be selected in MDK with a Configuration Wizard.



Traffic.c contains the code for the application. This code is based on tasks, which are scheduled by RTX.

The following source files are required for both bare metal and RTOS-based applications.

STM32F10x.s is the microcontroller start-up code, containing the exception vector table and reset handler.

STM32_Init.c is the microcontroller initialization code, which configures the memory and peripherals.

Retarget.c contains functions to retarget the C library. This file contains custom implementations of functions such as `fputc()`, so that `printf()` calls are directed to a serial communications port (UART).

LCD_4bit.c and Serial.c provide code to control the character LCD and the UART.

Getline.c includes a function to decode messages received via the UART.



Take 5 minutes to familiarize yourself with the code

RTX_Traffic Operation

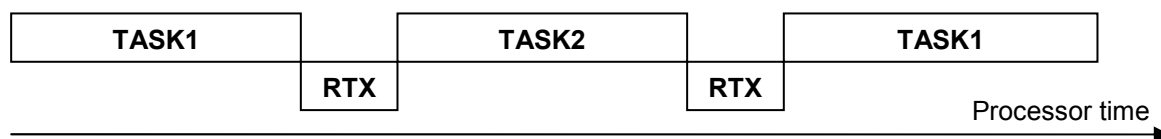
RTX_Traffic is representative of a real embedded system, in which a single controller can do a number of operations in parallel. Each of these operations needs to respond to external signals in a timely fashion.

The code on Traffic.c is split into tasks, which are executed concurrently. Each of these tasks handles one operation. You can see when a function is a task because of the way it is declared:

```
void init (void) __task { ... }
```

Customers interfacing a real-time system get the impression that all the tasks are running at the same time, but in reality the processor can only execute instructions from one task at a given point in time. The “trick” is done by the RTX kernel.

Every “timer tick” the system timer generates an interrupt and gives the control to RTX. RTX uses the timer tick to allocate time slots to different tasks. The timer tick is defined in RTX_Config.c and is 10,000us in the case of RTX_Traffic.



Because RTX allocates time to each task several times per second, they can respond to external events in a timely fashion. This way the system seems to do several operations at the same time.

RTX_Traffic Tasks

RTX_Traffic has up to 7 tasks executing in parallel.



Locate the code for each task in Traffic.c and match it with the description below

```
void clock (void) task { }
```

clock() is launched by RTX every second. It updates a clock variable and returns. By using RTX users do not need to configure the timers to generate an interrupt every second and do not have to modify the interrupt service routing to handle it.

```
void blinking (void) task { }
```

```
void lights (void) task { }
```

```
void lcd (void) task { }
```

blinking() implements “emergency traffic lights”. lights() implement “working traffic lights”. lcd() drives the MCBSTM32E character LCD.

The three tasks have the same requirement: to periodically update an output peripheral. In order to make the system efficient the peripheral is only updated when necessary. This is done easily with RTX thanks to the os_dly_wait() function, which implements a fixed delay between the instructions to set and clear individual traffic lights and to update the status of the LCD.

```
void get_escape (void) task { }
```

```
void keyread (void) task { }
```

get_escape() is periodically launched by RTX to check if an ESC command is received form the UART. Similarly keyread() is launched by RTX to check periodically the pedestrian push button.

By using RTX the peripheral is polled instead of using an interrupt. This has two advantages:

- It helps reduce the size of the interrupt handler and therefore make the system more responsive to external events
- The frequency at which the peripheral is polled is easily configured. This way the system responds to external signals “only” as fast as necessary, without wasting processor time

```
void command (void) task { }
```

command() communicates with the user via the UART. In essence this task is similar to keyread(), as it periodically polls the UART for new commands and then decodes them. The main difference is that command() does not “go to sleep” by using the os_dly_wait () function, but it tries to run as often as possible. This makes it the most responsive task in the system.

Build RTX_Traffic and Launch Simulation



Build Project using ‘Build Target’ button.

The example project builds with no errors and warnings.



Launch the simulator using ‘Start/Stop Debug Session’ button’

Task Initialization and Scheduling

Task Initialization

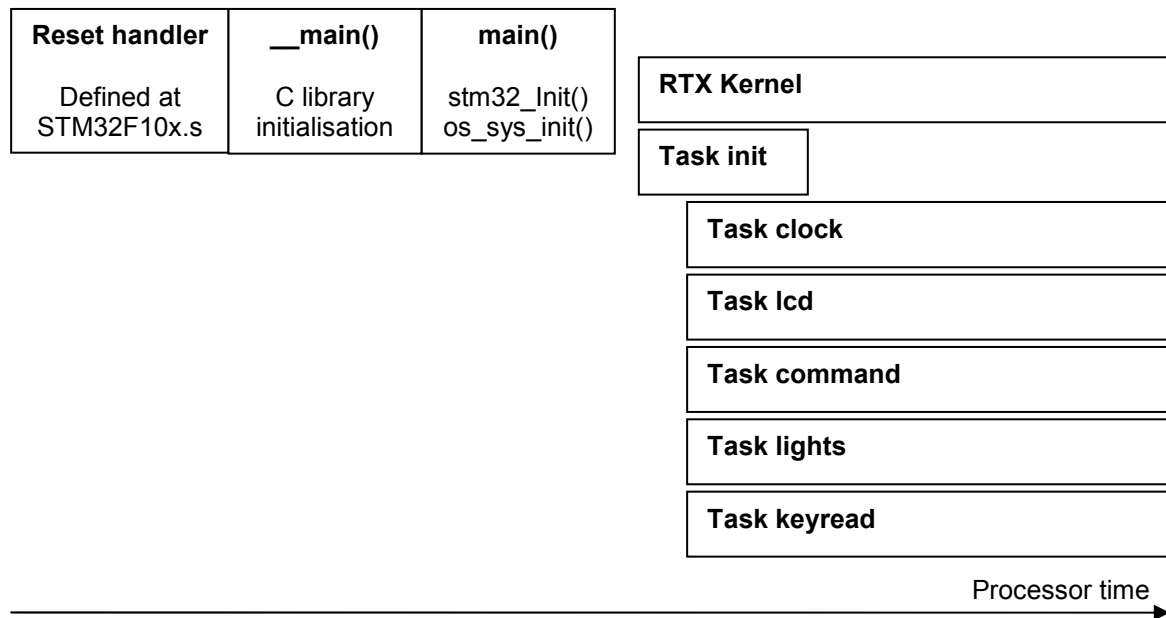
Since Traffic.c is structured in tasks, the main() function is only used to run the initialization code and launch RTX.

```
int main (void)    {
    stm32_Init ();
    os_sys_init (init); }
```

After booting, RTX launches the task `init`. This task launches all the other tasks and then stops.

```
void init (void) __task
{
    t_clock = os_tsk_create (clock, 0);
    t_lcd   = os_tsk_create (lcd, 0);
    t_command = os_tsk_create_user (command,0,&cmd_stack,sizeof(cmd_stack));
    t_lights = os_tsk_create (lights, 0);
    t_keyread = os_tsk_create (keyread,0);
    os_tsk_delete_self ();
}
```

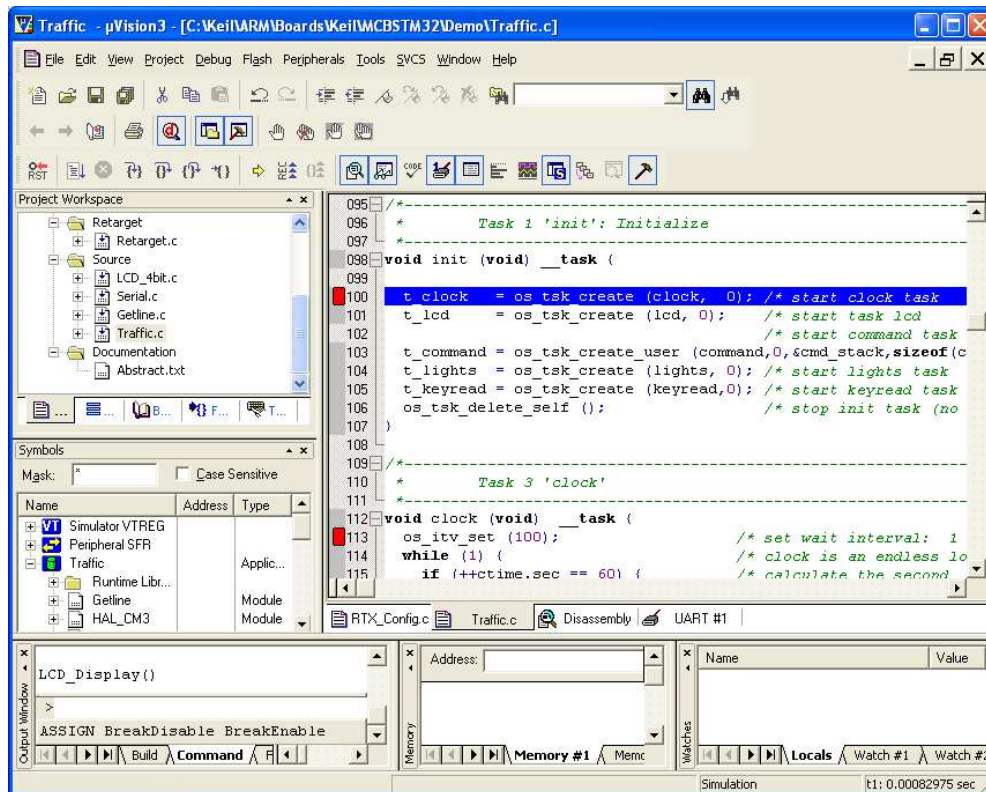
The following diagram shows how the different functions are executed in time.



Open Traffic.c



Double click on the left of the first line of the tasks to set a breakpoint
Do this for tasks init, clock, lcd, command, lights and keyread



Run Program

Repeat until simulation does not stop at a breakpoint

You can follow the sequence in which tasks are launched as they stop in the different breakpoints. This sequence is `init` → `clock` → `lcd` → `command` → `lights` → `keyread`.

After exiting the breakpoint each task goes into an infinite loop, implemented with a `while (1)` or `for(;;)` instruction:

```
void my_task (void) __task      {

    // Instructions that set up the task

    while (1) {

        // Instructions that are executed all over again
        // Optional RTX call to request "wake-up time"

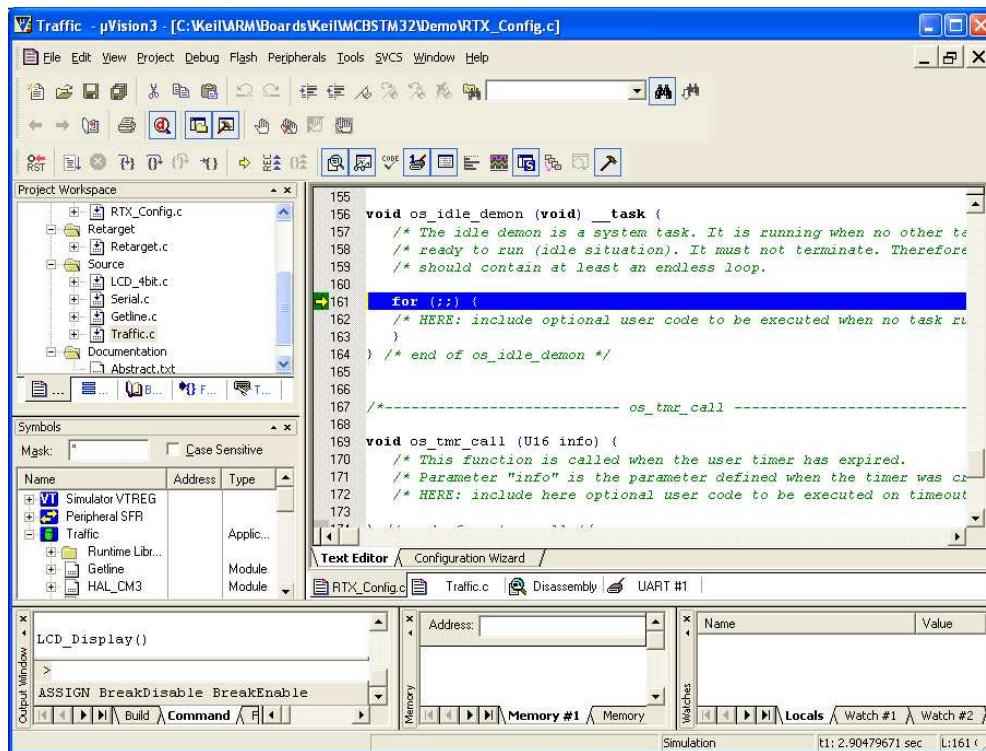
    }

}
```



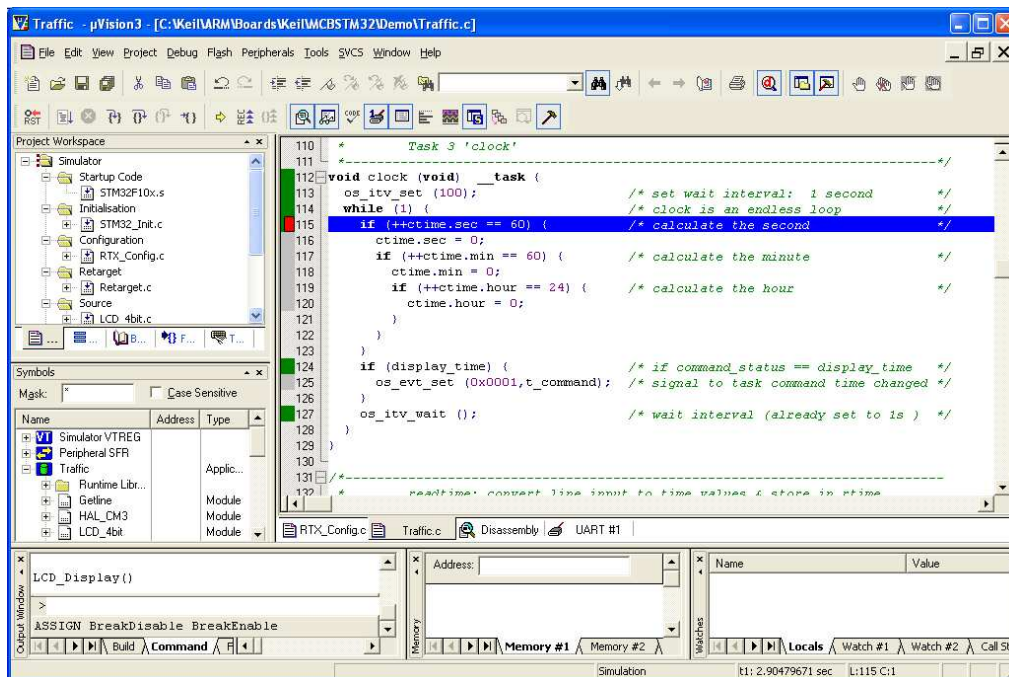
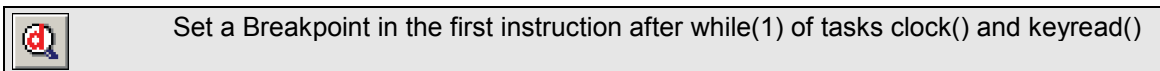
Halt Program

The chances are that the execution stops inside the `os_idle_demon()` task. This default task is part of RTX and runs if all the user tasks are sleeping.



Task Scheduling

This section explores how RTX calls different tasks periodically.



**Run Program**

Repeat as program stops at the breakpoints

You can see that RTX executes the task `keyread()` 20 times for each single execution of `clock()`.

The reason is that `clock()` runs every 100 system ticks...

```
Traffic.c: line 113      os_itv_set (100);          /* set wait interval: 1 second */
```

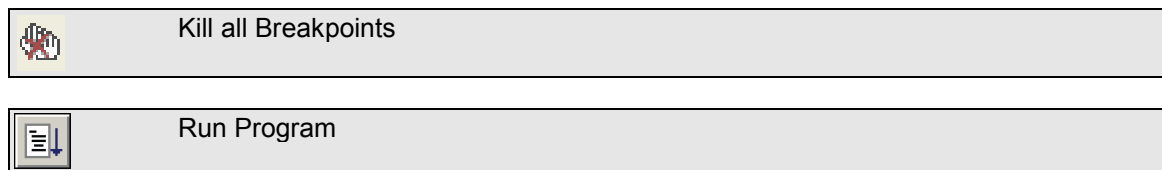
...while `keyread()` runs every 5 system ticks.

```
Traffic.c: line 314     os_dly_wait (5);          /* wait for timeout: 5 ticks */
```

**Kill all Breakpoints**

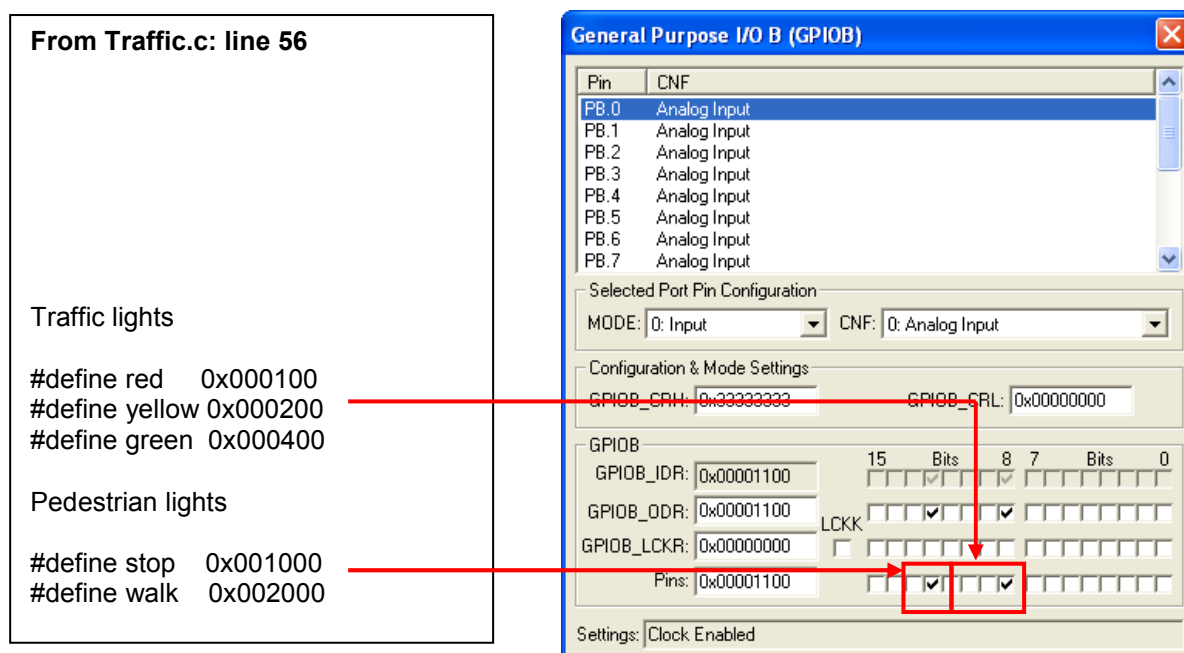
Analyzing the Code with μ Vision Debugger

This section shows how RTX is integrated in μ Vision Debugger.



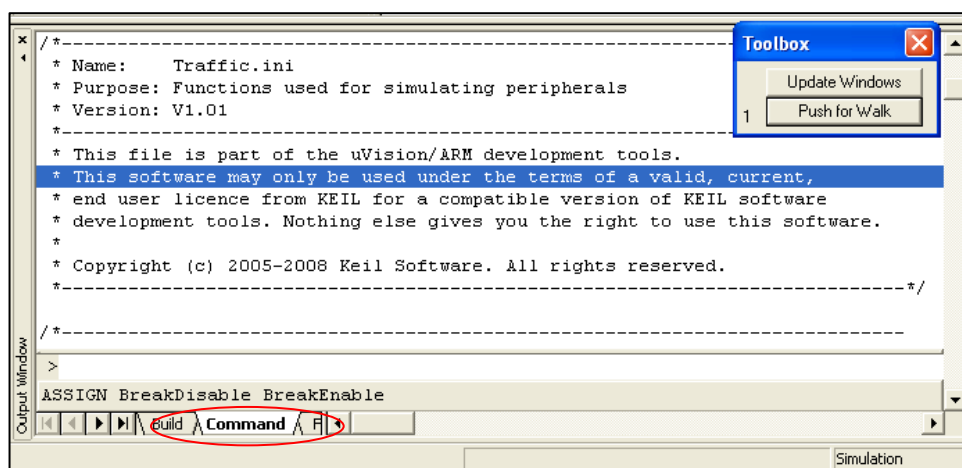
Peripheral Window

The Peripheral Window allows you to monitor the microcontroller GPIO, which are connected to outputs like the traffic lights (for cars) and a walk light (for pedestrians).



Toolbox

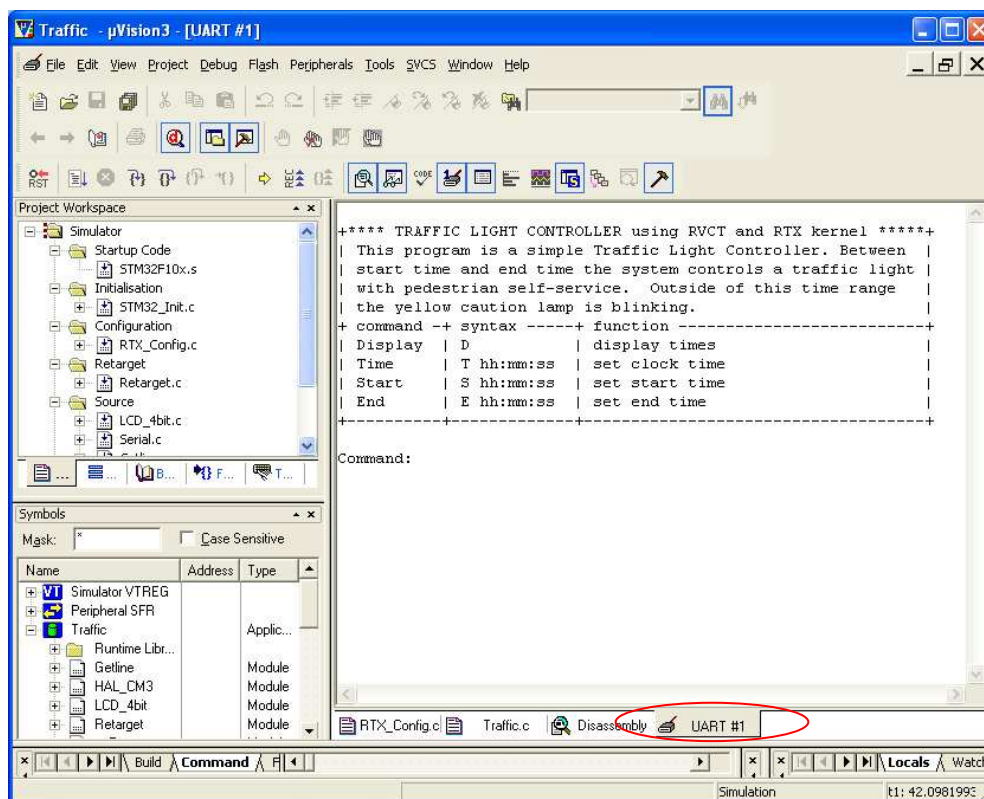
The file Traffic.ini contains a debugger script to open a Toolbox in simulation.



Use the "Push for Walk" button in the Toolbox to simulate a pedestrian wanting to cross. Watch the effect on the traffic lights in the Peripheral Window.

UART Window

Click on the UART #1 tab to communicate with the command() task.



RTX Kernel Analysis Window

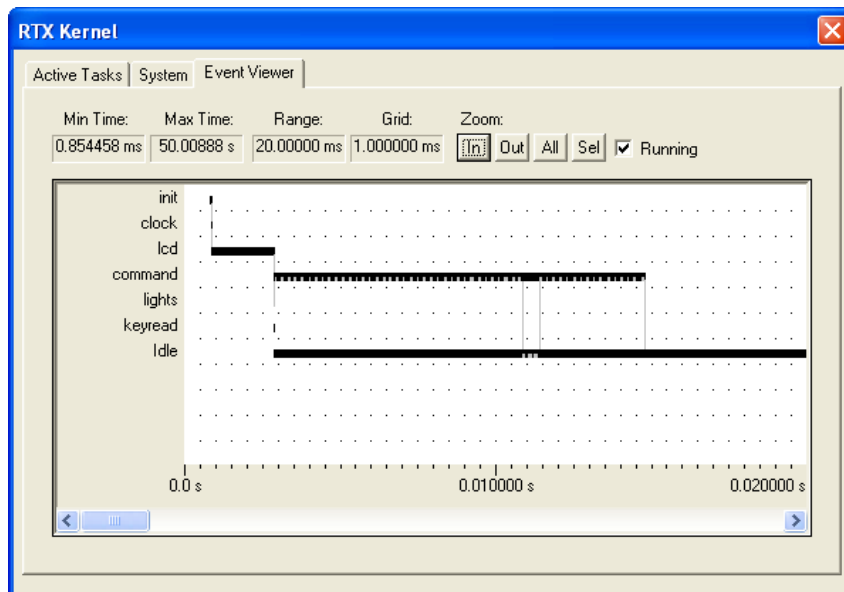


The Active Tasks tab shows information on the status of the tasks and its stack usage.

TID	Task Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
2	clock	1	WAIT_ITV	25			40%
3	lcd	1	WAIT_DLY	190			40%
4	command	1	WAIT_OR		0x0000	0x0100	19%
5	lights	1	WAIT_DLY	125			40%
6	keyread	1	WAIT_DLY	5			40%
255	os_idle_demon	0	RUNNING				0%

The System tab shows the RTX configuration and number of active and available tasks.

The Event Viewer tab shows the task execution in a time line. For example, by focusing the window on the start of the execution you can see when the different tasks are initialized.



Use all the resources provided by μ Vision Debugger to analyze the execution of RTX_Traffic.

Congratulations!

You have completed the RTX Real-Time Kernel Demonstration.

If you want to do some follow-up exercises, try the following:

Change the timer tick to 5,000us

- What is the effect on the program execution?
- What problems have appeared?
- What needs to be changed in the code to fix those problems?

Create a task that runs from the start and toggles a GPIO every 2s

- Can you see it in the RTX Event Viewer?

Connect with ULINK2 to an MCBSTM32E board and run the demo on real hardware